Lecture 8

More on MIPS, MIPS functions

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Administrivia

- Waitlist: Cleared soon!
- Concurrent Enrollment: We hear you!

- Thank you for feedback!

Agenda

- More on MIPS
- MIPS functions

Levels of Representation/Interpretation


```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

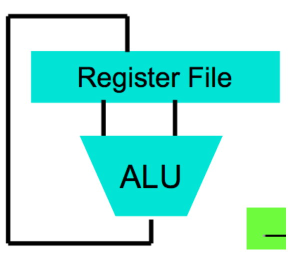
Anything can be represented as a *number*, i.e., data or instructions

Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

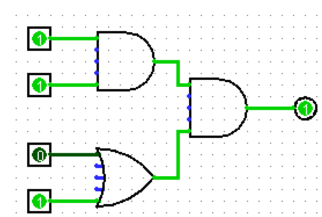
Machine Interpretation



Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit NOT	~	~	**`not`**
Shift left	<<	<<	**`sll`**
Shift right	>>	>>>	**`srl`**

Logic Shifting

- Shift Left: `sll $s1,$s2,2` #s1=s2<<2
  - Store in $s1 the value from $s2 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C.

Before: $0000\ 0002_{hex}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

After: $0000\ 0008_{hex}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1000_{two}$

What arithmetic effect does shift left have?

- Shift Right: `srl` is opposite shift; >>

Arithmetic Shifting

- Shift right arithmetic moves *n* bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$

- If executed sra $s0, $s0, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement

- MIPS: *if*-statement instruction is

```
beq register1,register2,L1
```

  means: go to statement labeled L1
  if (value in register1) == (value in register2)
  ....otherwise, go to next statement
- `beq` stands for *branch if equal*
- Other instruction: `bne` for *branch if not equal*

Types of Branches

- **Branch** – change of control flow

- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (`beq`) or branch *if not* equal (`bne`)

- **Unconditional Branch** – always branch
  - a MIPS instruction for this: *jump (`j`)*

Example *if* Statement

- Assuming translations below, compile *if* block

f → `$s0`   g → `$s1`   h → `$s2`

i → `$s3`   j → `$s4`

```
if (i == j)              bne $s3,$s4,Exit
  f = g + h;             add $s0,$s1,$s2
                 Exit:
```

- May need to negate branch condition!

Example *if-else* Statement

- Assuming translations below, compile

f → `$s0` g → `$s1` h → `$s2`

i → `$s3` j → `$s4`

```
if (i == j)          bne $s3,$s4,Else
  f = g + h;          add $s0,$s1,$s2
else                  j Exit
  f = g - h;  Else: sub $s0,$s1,$s2
              Exit:
```

Inequalities in MIPS

- Until now, we've only tested equalities

(== and != in C). General programs need to test < and > as well.

- Introduce MIPS Inequality Instruction:

  "Set on Less Than"

  Syntax: `slt reg1,reg2,reg3`

  Meaning: if (reg2 < reg3) reg1 = 1;
  else reg1 = 0;

  "set" means "change to 1",
  "reset" means "change to 0".

Inequalities in MIPS Cont.

- How do we use this? Compile by hand:
  if (g < h) goto Less; #g:$s0, h:$s1

- Answer: compiled MIPS code…

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$zero,Less  # if $t0!=0 goto Less
```

- Register $zero always contains the value 0, so bne and beq often use it for comparison after an slt instruction

- `sltu` treats registers as unsigned

Immediates in Inequalities

- `slti` an immediate version of `slt` to test against constants

```
Loop: . . .

slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1
beq  $t0,$zero,Loop   # goto Loop
                      # if $t0==0
                      # (if ($s0>=1))
```

Loops in C/Assembly

- Simple loop in C; A[] is an array of ints

```
do { g = g + A[i];
     i = i + j;
} while (i != h);
```

- Use this mapping: g, h, i, j, &A[0]
  $s1, $s2, $s3, $s4, $s5

```
Loop: sll  $t1,$s3,2      # $t1= 4*i
      addu $t1,$t1,$s5    # $t1=addr A+4i
      lw   $t1,0($t1)     # $t1=A[i]
      add  $s1,$s1,$t1    # g=g+A[i]
      addu $s3,$s3,$s4    # i=i+j
      bne  $s3,$s2,Loop   # goto Loop
                          # if i!=h
```

Agenda

- More on MIPS
- MIPS functions

Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and restore any registers you used
6. Return control to point of origin, since a function can be called from several points in a program

MIPS Function Call Conventions

- Registers faster than memory, so use them
- `$a0–$a3`: four *argument* registers to pass parameters ($4 - $7)
- `$v0,$v1`: two *value* registers to return values ($2,$3)
- `$ra`: one *return address* register to return to the point of origin ($31)

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Instruction Support for Functions (1/4)

C

```
 ... sum(a,b);... /* a,b:$s0,$s1 */
  }
     int sum(int x, int y) {
     return x+y;
  }
```

MIPS

```
address (shown in decimal)
  1000
  1004
  1008
  1012
  1016
  …
  2000
```

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
  int sum(int x, int y) {
  return x+y;
}
```

MIPS

```
address (shown in decimal)
 1000 add  $a0,$s0,$zero  # x = a
 1004 add  $a1,$s1,$zero  # y = b
 1008 addi $ra,$zero,1016 #$ra=1016
 1012 j    sum          #jump to sum
 1016 …                   # next instruction
 2000 sum: add $v0,$a0,$a1
 2004 jr   $ra       # new instr. "jump register"
```

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
return x+y;
}
```

MIPS

- Question: Why use `jr` here? Why not use `j`?
- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

```
2000 sum: add $v0,$a0,$a1
2004 jr   $ra   # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**`jal`**)
- Before:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum               #goto sum
```

- After:

```
1008 jal sum   # $ra=1012,goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common.
  - Don't have to know where code is in memory with **`jal`**!

MIPS Function Call Instructions

- Invoke function*: jump and link* instruction (`jal`)
  (really should be `laj` *"link and jump"*)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register `$ra`

```
jal FunctionLabel
```

- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register

```
jr $ra
```

Notes on Functions

- Calling program (*caller*) puts parameters into registers `$a0-$a3` and uses `jal X` to invoke (*callee*) at address labeled X
- Must have register in computer with address of currently executing instruction
  - Instead of *Instruction Address Register* (better name), historically called ***Program Counter*** (*PC*)
  - It's a program's counter; it doesn't count programs!

- What value does `jal X` place into `$ra`? **????**
- `jr $ra` puts address inside `$ra` back into PC

Where are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return, and delete
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- `$sp` is the *stack pointer* in MIPS ($29)
- Convention is grow from high to low addresses
  - *Push* decrements `$sp`, *Pop* increments `$sp`

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

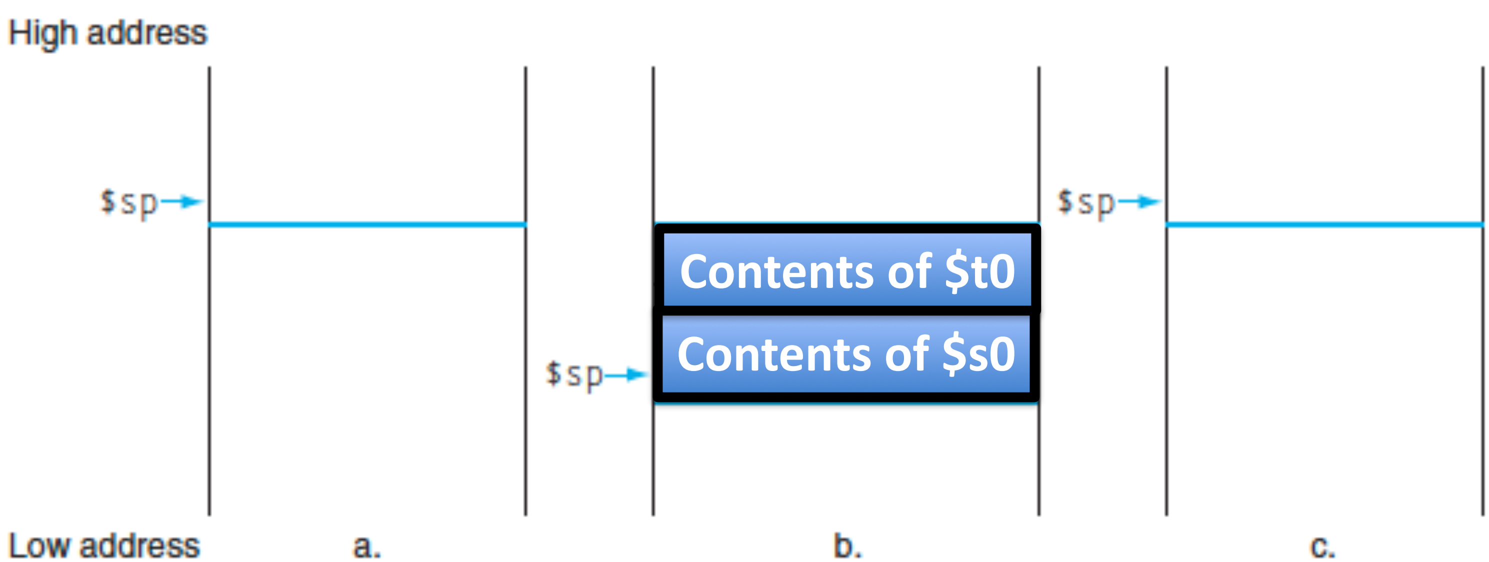
Example


```
int Leaf (int g, int h, int i, int j)
{
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Parameter variables `g`, `h`, `i`, and `j` in argument registers `$a0`, `$a1`, `$a2`, and `$a3`, and `f` in `$s0`
- Assume need one temporary register `$t0`

Stack Before, During, After Function

- Need to save old values of `$s0` and `$t0`



MIPS Code for Leaf()


```
Leaf: addi $sp,$sp,-8  # adjust stack for 2 items
      sw $t0, 4($sp)  # save $t0 for use afterwards
      sw $s0, 0($sp)  # save $s0 for use afterwards

      add $s0,$a0,$a1  # f = g + h
      add $t0,$a2,$a3  # t0 = i + j
      sub $v0,$s0,$t0  # return value (g + h) – (i + j)

      lw $s0, 0($sp)  # restore register $s0 for caller
      lw $t0, 4($sp)  # restore register $t0 for caller
      addi $sp,$sp,8  # adjust stack to delete 2 items
      jr $ra  # jump back to calling routine
```

What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in `$a0` to `$a3` and `$ra`
- What is the solution?

Nested Procedures (1/2)


```
int sumSquare(int x, int y) {
  return mult(x,x)+ y;
}
```

- Something called **sumSquare**, now **sumSquare** is calling **mult**
- So there's a value in $ra that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**

Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via **malloc**
  - Stack: Space to be used by procedure during execution; this is where we can save register values

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, MIPS divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - `$sp`, `$gp`, `$fp`, "saved registers" `$s0`- `$s7`
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Return value registers `$v0`,`$v1`, Argument registers `$a0`-`$a3`, "temporary registers" `$t0`-`$t9`, `$ra`

Clickers/Peer Instruction

- Which statement is FALSE?

A: MIPS uses `jal` to invoke a function and `jr` to return from a function

B: `jal` saves PC+1 in `$ra`

C: The callee can use temporary registers (`$ti`) without saving and restoring them

D: The caller can rely on save registers (`$si`) without fear of callee changing them